Automatic Synthesis of Control Software for an Industrial Automation Control System

Enrico Tronci[1]
Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, Coppito 67100 L'Aquila, Italy
tronci@univaq.it, http://univaq.it/~tronci

Abstract

We present a case study on automatic synthesis of control software from formal specifications for an industrial automation control system. Our aim is to compare the effectiveness (i.e. design effort and controller quality) of automatic controller synthesis from closed loop formal specifications with that of manual controller design followed by automatic verification.

Our experimental results show that for industrial automation control systems automatic synthesis is a viable and profitable (especially as far as design effort is concerned) alternative to manual design followed by automatic verification.

Key Words Formal Methods, Model Checking, Embedded Control Systems, Finite State Systems, Reactive Systems, Hybrid Systems, Discrete Event Systems, Supervisory Control, Manufacturing Systems.

1 Introduction

Industrial automation control systems are typical examples of safety critical *control systems.* A control system can often be partitioned into two main subsystems: a *controller* and a *plant.* Roughly speaking: (in an endless loop) the controller observes the state of the plant and on the base of such state information sends commands (stimulus) to the plant to achieve predefined goals.

The designer task is: given a plant model and closed loop requirements (i.e. the desired behaviour for the whole system, plant + controller) devise a suitable controller for the given plant.

To use formal methods (namely: model checking) to verify our controller (be it hardware or software) we need to write: formal specifications for the whole system (plant + controller), a finite state model of the plant, a program defining our controller (from which a finite state controller model is then obtained).

No matter how good our formal verification techniques are, testing and simulation will always be there. This means that, beyond defining our controller, a plant model will also be defined in order to run simulations. This even when formal methods are not used. Thus formal specs seem to be the price to pay to use automatic formal methods. A question then comes naturally. From a plant model and formal specs is it possible to automatically synthesize our embedded controller?

Using OBDDs [4] and model checking [7, 2] in [11, 12] it has been shown that when the plant can be modeled as a *Finite State System* (FSS) a (correct by construction) program (indeed a combinational circuit) implementing a controller can be automatically synthesized from the given formal specifications for the behaviour of closed loop system (plant + controller) and a plant (finite state) model. This saves on the coding effort for the controller and allows us to focus on formal specifications.

In this paper we present a case study applying the techniques developed in [11, 12]. Our goal is to automatically synthesize a controller for the production cell described in [8] from the closed loop specifications also given in [8].

The problem of verifying a manually designed controller for this production cell has been studied with various approaches (including model checking) in [8]. Interactive deductive synthesis from formal specifications has also been studied in [5]. However, as far as we know, no automatic synthesis from formal specifications has been attempted so far.

We want to assess viability of automatic controller synthesis from formal specifications for industrial problems of reasonable size. The (thoroughly studied) production cell case study presented in [8] well serves our purposes, since we can compare our design effort and controller quality with those reported in [8]. Moreover we can run our controller using the freely available Tcl/Tk graphic simulator provided by FZI [13].

Automatic synthesis of *supervisory controllers* has been studied in, e.g., [1, 3, 9]. For industrial automation control systems a case study on automatic synthesis of *supervisory controllers* has been presented in e.g. [3]. However for such systems no case study about automatic synthesis of *controllers* from formal specs has been previously presented. The results obtained from our case study can be summarized as follows.

- We succeeded in automatically synthesizing a controller for the given plant (production cell). This is a moderate size system with about 10^{12} states. Still, to the best of our knowledge, this is the largest system for which automatic controller *synthesis* has been attempted so far. Note however that automatic *verification* via model checking of systems even larger than our is routine.
- Our design time effort (1 man-week) compares favourably with the time needed to carry out manual design followed by automatic verification (ranging from weeks to months in the papers in [8]).

[1]This research has been partially supported by MURST funds.

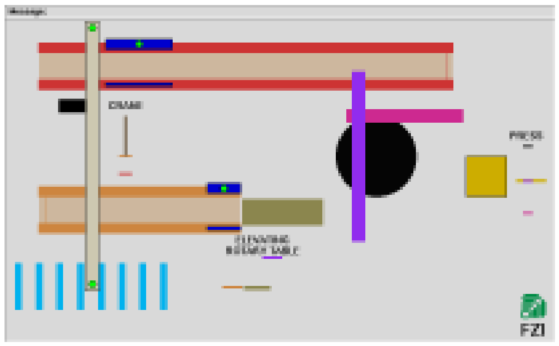

Figure 1: Production Cell (Tcl/Tk graphic simulator from FZI)

Command	Short Description
table_left	Starts rotation to the left of the ERT
table_stop_h	Stops rotation of the ERT
table_right	Starts rotation to the right of the ERT
table_upward	Starts upward movement of the ERT
table_stop_v	Stops vertical movement of the ERT
table_downward	Starts downward movement of the ERT

Figure 2: Elevating rotary table (ERT) commands

2 Production Cell Modeling

The production cell [8] is composed of two conveyor belts, an elevating rotary table, a two-armed robot, a press and a traveling crane. Figure 1 shows a top view of the production cell as it appears in the graphic simulator [13]. On the bottom [top] left is the feed [deposit] belt.

For simulation/validation purposes we used the Tcl/Tk graphic simulation environment for this plant freely available in [13].

The control program drives the plant by sending commands to the actuators and gets information about the status of the plant through sensors. E.g. the commands and sensors available for the elevting rotary table (ETR) are given respectively in figs. 2, 3. Note that the list of sensors also defines the (observable) state components of the plant.

Modeling Language

We use *First Order Logic* on a *Boolean* domain (BFOL) as our formal language.

This allows us to easily define transition relations as well as safety conditions and to use OBDDs [4] to carry out our computations.

Essentially BFOL can be seen as (formal) syntactic sugar for C based OBDD programming.

Our interpreter from BFOL to OBDD is called BSP (*Boolean Symbolic Programming*) [10]. It has a LISP like syntax that we will only show by examples.

A process (i.e. an FSS) p is represented with a boolean function (also named p) defining the transition relation of process p. We will denote with p`_px`, p`_nx`, p`_u` the arrays of boolean variables ranging, respectively, on p present states, p next states, p events (actions).

Short Description	Range/meaning
Is the ERT in the bottom position?	1=y/0=n
Is the ERT in the top position?	1=y/0=n
Angle of rotation of the ERT	$-5..90$

Figure 3: Elevating rotary table (ERT) sensors.

```
tabh [tabv] = table horizontal [vertical] postion;
tabl = table load (by blanket);
arm1 [arm2] = position robot arm 1 [2];
mag1 [mag2] = robot magnet of arm 1 [2];
crane_mag = crane magnet;
deposit = deposit belt;
deposit_blank = deposit belt photoelectric cell;
craneh [cranev] = crane horizontal [vertical] position;
crane_mag = crane magnet;
nop is an event leaving state unchanged;
robot state values are prefixed with rb_;
tabl_px, tabl_nx, tabl_u: present state, next state,
  event variables for process table_load.
```

Figure 4: Linking ids to informal requirements

```
(def sync (and sync_ft sync_tr sync_pr sync_dc))
(def unsafe_plant (and feedbelt feedbelt_load table_load
     magnet_1 magnet_2 arm1_load arm2_load press_load
     press_forge deposit deposit_blank deposit_load
     crane_load crane_mag tabv tabh arm1 arm2 robot press
     crane_hpos crane_vpos))
(def safex2xu_all (forall plant_all_nx (imp unsafe_plant
     (compose safe_x plant_all_px plant_all_nx))))
(def plant_all
     (and unsafe_plant safe_xu safex2xu_all sync))
(def end_all (and feedbelt_phaser table_phaser
     robot_phaser press_phaser deposit_phaser
     crane_phaser))
```

Figure 6: Plant and final states

The expression `(eqv` a b`)` is true iff the boolean arrays a and b are bitwise equal.

Cell Modeling

Potentiometers are used to measure displacements (robot arms) as well as angles (table, robot). Potentiometers return continuous values. However only a few milestone values are relevant for the controller. Since Analog to Digital (AD) conversion is done by taking this into account we can model the production cell using finite state automata.

Each cell component is built out of elementary components which, in turn, are all obtained by instatiating a generic process model. E.g. the elevating rotary table is described by two processes: `tabh` and `tabv` modeling respectively the horizontal and vertical motion of the table.

Each cell component works accordingly to a fixed sequence of operations (phases). This behaviour is given in the requirements in [8] thus it will be part of our plant model. For process p this is modeled by function p`_phaser`.

As a result the controller task essentially is to decide for each cell component when to switch from one phase to the next one. This must be done optimizing performance without violating the given safety requirements.

Plant Model

Plant components exchange blanks. E.g. the rotary table gets a blank from the feed belt, the robot gets a blank from the rotary table as well as from the press, etc. Blank exchange can be modeled as process communication via suitable synchronization constraints (`sync_`...) saying when blank exchange is possible and what happens upon a blank exchange. E.g. function

Plant state bits	Plant state space size	CPU time (min)	Max OBDD	Ctr code size (bytes)	Ctr exe size (bytes)	Ctr min rct t (ms)	Ctr max rct t (ms)	Ctr avg rct t (ms)
42	$4.4*10^{12}$	15:14	1,851,657	202,660	197,248	0.0	10.0	0.6

Figure 5: Experimental Results on Linux PC with 200 MHz Pentium and 112MB of RAM

sync_ft models synchronization between the feed belt and the rotary table.

Function safe_xu [safe_x] defines the set of transitions [states] that are considered safe accordingly to the in the requirements given in [8].

The transition relation for the plant process plant_all is given in fig. 6, where unsafe_plant is the transition relation for our production cell.

We need to restrict (logical *and*) unsafe_plant behaviour to transitions that are safe (defined with safe_xu) and that may only lead to a safe state (defined with safex2xu_all in fig. 6).

In function safex2xu_all in fig. 6 (compose safe_x plant_all_px plant_all_nx) is obtained from safe_x by simultaneously replacing plant_all_px with plant_all_nx.

At last we require that synchronization (sync in fig. 6) be satisfied. This leads us to the definition for plant_all in fig. 6.

Final States
The set of final (goal) states is the set of states that the controller strives to reach. It is defined with its characteristic function (end_all in fig. 6) which is simply the logical *and* of the phase functions.

The set of goal states end_all models the liveness requirements in [8]. In fact the synchronization (sync in fig. 6) between cell components is such that when a cell component reaches a certain phase a transition is triggered making the phase function for that cell component true only on states belonging to the next phase. As a result the controller will drive each cell component endlessly from one phase to the next one.

3 Experimental Results

From the plant model plant_all and the set of final states end_all defined in fig. 6 we can automatically synthesize a (correct by construction) C program implementing a (centralized) embedded controller $\mathcal{K}$ for our plant. This is done as shown in [11, 12]. Fig. 5 gives our experimental results. Note that columns "Plant state ..." in fig. 5 refer to the plant state space. The controller itself is as a combinational circuit (implemented in C in our case) with 42 inputs (representing the plant present state) and 46 outputs (representing plant events). Column "CPU" gives the CPU time for synthesizing our controller starting from formal specs. Column "Max OBDD" is the largest OBDD created during the controller synthesis computation. Columns "Ctr..." reports the synthesized controller code size, executable size and max, min, average reaction time (i.e. the time our controller takes to compute a plant command from a plant state sample given in input).

It took us about 1 man-week to write formal specs and plant model. One more man-week was needed to write the interface to the Tcl/Tk simulator provided by FZI [8].

Fig. 7 shows effort data for the case studies in [8] for which such data where available (NA stands for "not available"). The first column identifies the case study using the name of the language or tool used to carry it out. The row labeled "Automatic Synthesis" gives data for our approach. Column "FSV" gives data on Formal Specification and Verification. Columns "Ctr time", "Ctr src", "Ctr exe" give data for, respectively, the controller development time, source code, executable code. Columns "I time", "I src" give data for, respectively, development time and source code for the interface to the FZI simulator. Column "blk" gives the maximum number of metal blanks simultaneously present in the production cell that can be correctly handled by the controller.

It must be kept in mind that the case studies in [8] were developed by teams with different levels of expertise (some where developed by students some by experienced researchers). Moreover the tools used were also at different stages of development. Thus the data in fig. 7 can only be considered as a rather rough indication. Anyway, these are the only data we have.

Once a plant formal model is available synthesizing a C program implementing our embedded controller takes about 15 minutes on a Linux PC with a 200 MHz Pentium and 112MB RAM. Our controller C code size as well as executable size (fig. 5) are reasonable, although not particularly small. E.g. the controller executables obtained via manual design using Esterel or Signal have size, respectively, 46K and 90K (fig. 7).

Reaction times in fig. 5 where measured by running our controller on our Linux PC and inserting calls to the clock() C function in our controller code. We found no data about reactions times in [8]. Anyway all controllers in [8] as well as our are "fast enough" for the plant production cell accordingly to the FZI simulator.

Our controller works for all possible relative speeds of cell motors and can handle up to 6 metal blanks simultaneously present in the production cell. This is the same or more than many of the manually designed controllers in [8] can do. Note however that, by suitably refining requirements, the controller designed in [6] (TLT in fig. 7) can handle up to 8 metal blanks in the system.

Altogether from this case study we conclude that for moderate size industrial automation plants automatic embedded controller synthesis from closed loop formal specifications compares favourably with manual design followed by automatic verification. (ranging from weeks to months in fig. 7).

Indeed if we consider that to carry out automatic formal verification (model checking) we need: a plant finite state model, closed loop formal specs *and* a controller finite state model whereas only the first two items are needed when using automatic synthesis (AS) is it clear that AS (when usable) saves on design effort.

On the other hand when using AS the controller executable size is larger that that of a manually designed controller. We are not able to compare reaction times. However, because of the structure of the synthesized C

	FSV	Ctr time	Ctr src	Ctr exe	I time	I src	blk
CSL	3 weeks for specification and verification via model checking	3 days	9 pages (CSL)	NA	1 week	7 pages (C)	4
ESTEREL	Only specification, no verification.	NA	NA	46KB	NA	NA	NA
LUSTRE	About 1 week for specification and verification using Lesar	$\approx$ 1 week	200 lines (LUSTRE) yielding 800 lines (C)	NA	$\approx$ 2 weeks	NA	5
SIGNAL	1 week for specification and partial verification using Sigali	2 weeks	1700 lines (C)	90KB	1 week	NA	NA
StateCharts	About 1 week for specification and verification via model checking	$\approx$ 2 weeks	NA	NA	NA	NA	5
TLT	NA	$\approx$ 2 weeks	NA	NA	NA	NA	8
SDL	About 1 month for specification and testing using SDT, no verification.	1 month	1800 lines (SDL92)	NA	NA	NA	NA
Tatzelwurm	About 600 lines for specification, verification done using interactive proofs.	2 weeks	250 lines (Pascal)	NA	NA	NA	6
HTTDs and HOL	About 2 months for specification and verification using interactive proofs.	NA	NA	NA	NA	NA	NA
Deductive Synthesis	More than 1 month for carrying out proofs yielding a controller.	0 days	NA	NA	NA	NA	NA
Automatic Synthesis	1 week for specification. No verification needed.	0 days	203K	197K	1 week	65K	6

Figure 7: Comparing efforts from [8] with our Automatic Synthesis approach.

code, we think this should not be a problem, at least for industrial automation problems (low bandwidth).

Note that our approach only makes sense when writing closed loop formal specs is easier than manually designing our system and then verifying it. This is often the case for embedded control systems, but it is false in general.

4 Conclusions

Our results (sec. 3) show that, for moderate size industrial automation control systems, automatic synthesis of embedded controllers from formal specifications is a viable and indeed profitable alternative to manual design followed by automatic verification.

We see two main obstructions to our approach: state explosion (also present in automatic verification via model checking); controller size (is larger than that obtained by using manual design, for some application this could be a problem). These obstructions are obvious subjects for our future research.

References

[1] S. P. Khatri, A. Narayan, S. C. Krishnan, K. L. McMillan, R. K. Brayton, A. Sangiovanni-Vincentelli *Engineering Change in a Non-Deterministic FSM Setting*, Proc. of 33rd IEEE/ACM "Design Automation Conference", 1996

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, *Symbolic model checking: 10^{20} states and beyond*, Information and Computation 98, (1992)

[3] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, G. F. Franklin, *Supervisory Control of a Rapid Thermal Multiprocessor*, IEEE Trans. on Automatic Control, Vol. 38, N. 7, July 1993

[4] R. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. on C., Vol. C-35, N.8, Aug. 1986

[5] J. Burghardt, *Deductive Synthesis*, in [8]

[6] J. Cuellar, M. Huber *TLT*, in [8]

[7] E. A. Emerson, *Temporal and Modal Logic*, Handbook of Theoretical Computer Science, Elsevier 1990

[8] C. Lewerentz, T. Lindner (Eds), *Formal Development of Reactive Systems: Case Study Production Cell*, LNCS 891, Springer-Verlag 1995

[9] P. J. Ramadge, W. M. Wonham, *The Control of Discrete Event Systems*, Proc. of the IEEE, Jan. 1989

[10] E. Tronci, *Hardware Verification, Boolean Logic Programming, Boolean Functional Programming*, Proc. 10th IEEE Conf. on "Logic In Computer Science" 1995, San Diego, CA, USA

[11] E. Tronci, *On Computing Optimal Controllers for Finite State Systems*, Proc. of 36th IEEE CDC, 1997, USA

[12] E. Tronci, *Automatic Synthesis of Controllers from Formal Specifications*, Proc. of 2nd IEEE Int. Conf. on "Formal Engineering Methods", Dec. 1998, Brisbane, Australia

[13] Ftp to `gate.fzi.de`, login as `anonymous`, cd to `pub/korso/fzelle/simulation` get file `visualization.tar.Z`